Synthèse d'Image - Lancer de rayon - TD01

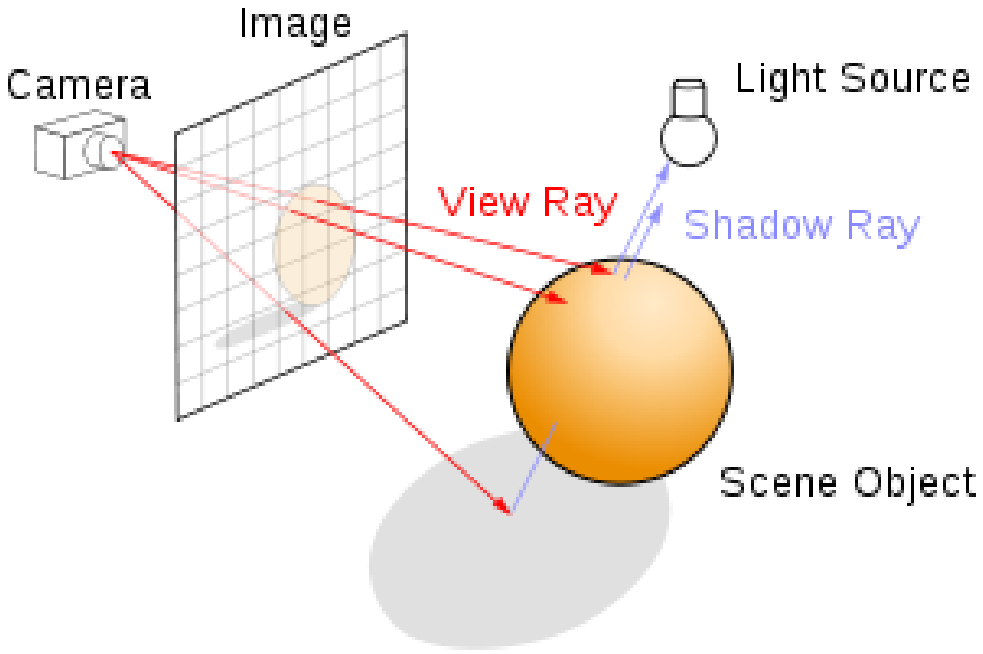
Introduction au Raytracing

L'objectif de ce TD est de comprendre les fondamentaux du Lancer de Rayon (Ray-tracing) et de développer une petite librairie mathématique qui sera utilisée par la suite lors de la réalisation d'un véritable de lancer de rayon !

Introduction

De manière générale, le **Lancer de rayon** est une technique qui consiste à suivre les rayons de lumière qui se propagent dans la scène et qui arrivent sur la caméra. Ce sont ces rayons de lumière qui transportent la couleur des objets.

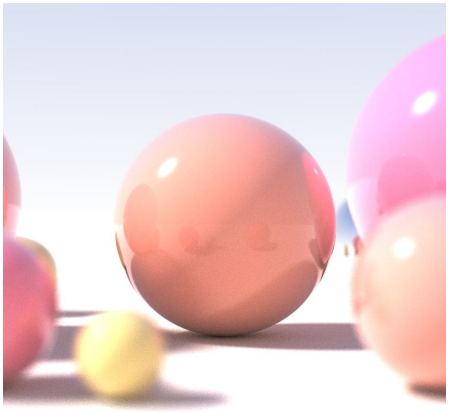
Il existe plusieurs algorithmes de lancer de rayon : le *backward raytracing*, le *forward raytracing,* le *bidirectional raytracing*, etc. Dans cette série de TD, nous implémenterons le plus simple : le **backward raytracing** :



Le principe de cet algorithme est de retracer le chemin de la lumière à partir de la caméra. Pour chaque pixel de l'image finale, un (ou plusieurs) rayon(s) sont lancés jusqu'à intersecter un objet de la scène. Lorsqu'une intersection est détectée, le rayon qui a intersecté l'objet rebondit dans une autre direction (qui dépend des caractéristiques de la surface de l'objet). A chaque rebond, le rayon enregistre la quantité de lumière et la couleur de la lumière correspondant à la surface de l'objet intersecté. Le rayon s'arrête lorsqu'il rencontre une source de lumière.

Le lancer de rayon est donc un algorithme fondamentalement récursif : on lance un rayon, qui lance lui même des rayons, qui peuvent eux même lancer des rayons, etc. Les couleurs s'accumulent le long de ces rayons selon les lois de réflection et de réfraction de la physique des surfaces et des matériaux rencontrés.

Voici le niveau de réalisme qu'on peut obtenir avec de bons raytracers (et beaucoup de temps de calcul) :



Différence avec les techniques de "Rasterisation"

Les techniques de lancer de rayon servent le même propos que les techniques dites de "rasterisation" (OpenGL, DirectX, etc.), à savoir créer des images. Cependant, elles ne sont pas utilisées dans le même type de programmes.

Les techniques de Rasterisation sont principalement utilisées pour calculer des images en temps réel, chose que ne permet pas les techniques de lancer de rayon car trop coûteuses en temps. De ce fait, elles sont massivement utilisées par exemple dans le jeux vidéo, dans des applications 3D sur internet (WebGL), ou dans des logiciels de conception 3D. Elles utilisent principalement les ressources des cartes graphiques et s'articulent sur l'architecture technique de ces dernières pour calculer la couleur finale des pixels.

Les techniques de lancer de rayon sont généralement plus coûteuses en temps de calcul, et donc utilisées pour calculer des images fixes. En contrepartie, elles peuvent produire des images très proches de la réalité en utilisant des algorithmes basés sur la physique du monde réel. Elles sont principalement utilisées lors de la réalisation de films d'animation, mais aussi pour calculer des images d'architecture (intérieur de maison), et même pour le catalogue Ikea (75% des "photos" du catalogue Ikea en 2016 étaient en fait des images de synthèse selon le site co.design).

EXERCICE 01 : Mise en place de la structure du programme

Nous allons mettre en place la structure de notre raytracer. Vous utiliserez le même programme pour tous les TDs relatifs au lancer de rayon.

1. **Créez un dossier nommé *raytracer/* (au même niveau que vos dossiers TD01, TD02, etc. par exemple).**
2. **Dans le dossier *raytracer/*, créez les dossiers suivants :**
    a. **le dossier *include/* qui contiendra les headers de votre programme**
    b. **le dossier *src/* qui contiendra le code du raytracer et l'implémentation de vos différentes fonctions.**
    c. **le dossier *obj/* qui contiendra les objets issus de la compilation du programme (les fichiers .o)**
    d. **le dossier *bin/* qui contiendra l'exécutable de votre programme**
3. **Dans le dossier *raytracer/src/*, créez un fichier *main.c*. Copiez y le code suivant :**

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv) {
    return EXIT_SUCCESS;
}
```

4. **Dans le dossier *raytracer/*, créez un fichier Makefile. Copiez y le code suivant :**

```
CC = gcc
CFLAGS = -Wall -O2 -g
LDFLAGS = -lSDL -lm

APP_BIN = raytracer

SRC_PATH = src
OBJ_PATH = obj
INC_PATH = -I include
BIN_PATH = bin
LIB_PATH = lib

SRC_FILES = $(shell find $(SRC_PATH) -type f -name '*.c')
OBJ_FILES = $(patsubst $(SRC_PATH)/%.c,$(OBJ_PATH)/%.o, $(SRC_FILES))

all: $(APP_BIN)

$(APP_BIN): $(OBJ_FILES)
        @mkdir -p $(BIN_PATH)
        $(CC) -o $(BIN_PATH)/$(APP_BIN) $(OBJ_FILES) $(LDFLAGS)

$(OBJ_PATH)/%.o: $(SRC_PATH)/%.c
        @mkdir -p "$(@D)"
        $(CC) -c $< -o $@ $(CFLAGS) $(INC_PATH)

clean:
        rm $(OBJ_FILES) $(BIN_PATH)/$(APP_BIN)
```

EXERCICE 02 : Vecteurs et Points

Pour développer notre raytracer, nous aurons besoin d'objets et de fonctions mathématiques. Parmis les objets mathématiques très utilisés en synthèse d'image, on trouve le Point et le Vecteur. Nous allons créer des structures et des fonctions utilitaires permettant de manipuler ces identités.

1. **Créez un fichier *geometry.h* dans le dossier *include/*, en prenant soin de définir des variables propres à ce header pour qu'il ne soit pas réimporté à chaque instruction #include.**

2. **Créez une structure `Vec3` contenant trois champs de type float appelés x, y et z.**

3. **Déclarez deux nouveaux types de variables : `Point3D` et `Vector3D`, qui sont en fait des alias vers le type struct `Vec3`. Nous les utiliserons pour différencier ces deux types de données et ainsi éviter les ambiguïtés.**

4. **Dans votre fichier *geometry.h*, ajoutez les signatures des fonctions suivantes :**

   `Point3D pointXYZ(float x, float y, float z)`
   → Construit le point (x, y, z)
   `Vector3D vectorXYZ(float x, float y, float z)`
   → Construit le vecteur (x, y, z)
   `Vector3D vector(Point3D A, Point3D B)`
   → Construit le vecteur $\vec{AB}$ = B − A
   `Point3D pointPlusVector(Point3D P, Vector3D V)`
   → Construit le point P + V
   `Vector3D addVectors(Vector3D A, Vector3D B) et subVectors(...)`
   → fonctions d'addition et soustraction de vecteurs
   `Vector3D multVector(Vector3D V, float a) et divVector(...)`
   → fonctions de multiplication et division d'un vecteur par un scalaire
   `float dot(Vector3D A, Vector3D B)`
   → Fonction calculant le produit scalaire de deux vecteurs
   `float norm(Vector3D A)`
   → fonction calculant la norme d'un vecteur
   `Vector3D normalize(Vector3D A)`
   → fonction retournant le vecteur normalisé passé en paramètre

5. **Créez un fichier *geometry.c* dans le dossier *src/* et implémentez y les fonctions définies ci dessus.**

6. **Rappelez ce qu'est un produit scalaire entre deux vecteurs (= ce qu'il représente en termes mathématiques).**

EXERCICE 03 : Un peu de couleur

Les rayons de notre raytracer devront transporter et accumuler de la couleur. Nous allons donc définir des structures et des fonctions utilitaires pour manipuler des couleurs.

1. **Créez un fichier *colors.h* dans le dossier *include/*, en prenant soin de définir des variables propres à ce header pour qu'il ne soit pas réimporté à chaque instruction #include.**

2. **Créez une structure Color3f contenant 3 champs de type float appelés r, g et b.**
   Dans cette structure, nous représenterons chaque canal par une valeur comprise entre 0.0 et 1.0. A terme, lorsque nous afficherons des pixels sur une image à l'écran, il faudra reconvertir cet espace de couleur dans un espace approprié propre à la librairie d'affichage.

3. **Dans votre fichier *colors.h*, ajoutez les signatures des fonctions suivantes :**

   `Color3f addColors(Color3f c1, Color3f c2)`
   → Fonction qui calcule la somme de deux couleurs
   `subColors, multColors (même signature)`
   → L'équivalent pour la soustraction et la multiplication.
   `Color3f multColor(Color3f c, float a) et Color3f divColor(...)`
   → fonctions de multiplication et division d'une couleur par un scalaire

4. **Créez un fichier *colors.c* dans le dossier *src/* et implémentez y les fonctions définies ci dessus.**

EXERCICE 04 : Testons nos fonctions

Etant donné que cette base de code va nous resservir pour tout le programme, il convient de la tester correctement.

1. **Dans le fichier *geometry.h*, créez une fonction `void printVector3D(Vector3D v)` et une fonction `void printPoint3D(Point3D p)` qui affichent les contenus du vecteur et du point passés en paramètre.**

2. **Dans le fichier *main.c*, vérifiez les égalités suivantes en utilisant les fonctions définies aux exercices précédents :**

Opération	Résultat

pointPlusVector : (0, 0, 0) + (1, 2, 0)	(1, 2, 0)
addVectors : (0.5, 1.0, -2.0) + (0.2, -1.0, 0)	(0.7, 0.0, -2.0)
subVectors : (0.5, 1.0, -2.0) - (0.2, -1.0, 0)	(0.3, 2.0, -2.0)
multVector : (0.5, 1.0, -2.0) * 2.0	(1.0, 2.0, -4.0)
multVector : (0.5, 1.0, -2.0) * 0.0	(0.0, 0.0, 0.0)
divVector : (0.5, 1.0, -2.0) / 2.0	(0.25, 0.5, -1.0)
divVector : (0.5, 1.0, -2.0) / 0.0	?
dot((1.0, 0.0, 0.0), (2.0, 0.0, 0.0))	2.0
dot((1.0, 0.0, 0.0), (0.0, 1.0, 0.0))	0
norm((2, 0, 0)	2.0
norm((1, 1, 1)	1.732051
norm((0, 0, 0))	0
normalize((1, 1, 1))	(0.57735, 0.57735 0.57735)
normalize((0, 0, 0))	?

EXERCICE 05 (à faire à la maison) : Intersections

Etant donné :

- Un objet Rayon, définit par un point O (x, y, z) et une direction D (x, y, z)
- Une objet Sphère, définit par un point O (x, y, z) et un rayon r
- Un objet Cube, définit par les deux points de sa diagonale : Pmin (x, y, z) et Pmax (x, y, z).

1. **Trouvez l'équation d'intersection entre un rayon et une sphère.**
2. **Trouvez l'équation d'intersection entre un rayon et un cube.**

Ecrivez le pseudo code de ces équations sans les implémenter (au prochain TP).

Pour vous aider, vous pouvez vous poser les questions suivantes :

- A quelle condition un point P(x, y, z) se trouve sur un rayon ?
- A quelle condition un point P(x, y, z) se trouve sur une sphère ?
- A quelle condition un point P(x, y, z) se trouve sur un cube ?